Towards the Introduction of QoS Information in a Component Model

Sun Meng
Centrum Wiskunde & Informatica (CWI)
P.O. Box 94079
Amsterdam, The Netherlands
M.Sun@cwi.nl

Luis S. Barbosa
Department of Informatics
Minho University
Braga, Portugal
lsb@di.uminho.pt

ABSTRACT

Assuring Quality of Service (QoS) properties is critical in the development of component-based distributed systems. This paper presents an approach to introduce QoS constraints into a coalgebraic model of software components. Such constraints are formally captured through the concept of a Q-algebra which, in its turn, can be smoothly integrated in the definition of component combinators.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Requirements/Specifications

Keywords

Component-based design, coalgebra, QoS properties

1. INTRODUCTION

Often software services execute on the hardware of their respective providers, in different containers, separated by firewalls and other trust barriers. Actually, the design of loosely-coupled, highly distributed software systems places new requirements on service and components' composition. To build applications, one often needs to select them from a set of functionally equivalent candidates, *i.e.*, components that implement the same functionality but differ in their non-functional characteristics, i.e., *Quality of Service* (QoS) properties [6, 8]. QoS properties of individual components, such as response time, availability, bandwidth requirement, memory usage, etc., cannot be ignored and become decisive in the selection procedure. Moreover, often adaptation mechanisms have to take them into account, going far behind simple functionality wrapping to bridge between published interfaces.

Dealing with QoS aspects in a coherent and systematic way became a main issue in component composition, which cannot be swept under the carpet in any formal account of the problem.

This paper suggests how a formal calculus for component composition [2, 3] can be extended in order to take into account, in an explicit way, QoS information. The calculus is based on a coalgebraic model used to capture components' observable behavior and persistence over transitions. Furthermore, it is parametric on a notion of *behavior*, encoded in a strong monad, which allows to reason in a uniform way about total or partial, non deterministic or stochastic components. It has already been argued by others (e.g., [5, 7]) that coalgebra theory nicely captures the "black-box" characterization of software components, which favors an *observational* semantics: the essence of a component specification lies in the collection of *possible observations* and any two internal configurations should be identified wherever indistinguishable by observation.

The notion of Q-algebra proposed in [4] is adopted to express QoS properties. In brief, a Q-algebra amounts to two semirings over a common carrier, representing some form of *cost* domain, which allows different ways of combining and choosing between quality values. The resulting calculus provides a compositional approach which offers potential for complex components to be constructed systematically while satisfying QoS constraints.

2. COMPONENTS AS COALGEBRAS

Software components can be characterized as dynamic systems with a public interface and a private, encapsulated state. The relevance of state information precludes a 'process-like' (purely behavioral) view of components. Components are rather *concrete* coalgebras [7, 1, 5]). For a given value of the state space — referred to as a *seed* in the sequel — a corresponding 'process', or *behavior*, arises by computing its coinductive extension. Such a coalgebraic model provides an observational semantics for software components and a generic assembly calculus where the behavior pattern of a component is abstracted to a strong monad B. For example, $\mathsf{B} = \mathsf{Id}$ retrieves the simple deterministic behavior, whereas $\mathsf{B} = \mathcal{P}$ or $\mathsf{B} = \mathsf{Id} + \mathbf{1}$ would model non deterministic or partial behavior, respectively.

Assume a collection of sets I, O, ..., acting as component interfaces. Then a component taking input in I and producing output in O is specified by a pointed coalgebra

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle \qquad (1)$$

for functor $\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)^I$, where u_p is the initial state, often referred to as the *seed*, and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$. This definition means that the computation of an action in a component will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations.

Having defined generic components as (pointed) coalgebras, one may wonder how do they get composed and what kind of calculus emerges from this framework. In this framework, interfaces are sets representing the input and output range of a component. Con-

SAC'10 March 22-26, 2010, Sierre, Switzerland.

sequently, components are arrows between interfaces and so arrows between components are arrows between arrows. Thus, three notions have to be taken into account: interfaces, components and component morphisms. A component morphism $h : \langle u_p, \overline{a}_p\rangle \longrightarrow \langle u_q, \overline{a}_q\rangle$ is just a function connecting the state spaces of p and q and satisfying the following *morphism* and *seed preservation* conditions:

$$\overline{a}_q \cdot h \;=\; \mathsf{T}^{\mathsf{B}}\, h \cdot \overline{a}_p \tag{2}$$

$$h\; u_p \;=\; u_q \tag{3}$$

Components with compatible interfaces (as in the case $p : I \longrightarrow K$ and $q : K \longrightarrow O$) can be composed sequentially as

$$p\,;q \;=\; \langle \langle u_p, u_q\rangle \in U_p \times U_q, \overline{a}_{p;q}\rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows

$$\begin{aligned} a_{p;q} \;= & U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \\ & \mathsf{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(\mathsf{a}\cdot\mathsf{xr})} \\ & \mathsf{B}(U_p \times (U_q \times K)) \xrightarrow{\mathsf{B}(\mathsf{id}\times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \\ & \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \xrightarrow{\mathsf{BBa}^\circ} \\ & \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

There are a collection of component combinators cater component aggregation. For example, *external choice* $\boxplus$ and *parallel* $\boxtimes$ composition. When interacting with $p \boxplus q : I + J \to O + R$, the environment chooses either to input a value of type I or one of type J, which triggers the corresponding component (p or q, respectively), producing the relevant output. In its turn, parallel composition corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavioral effect, captured by monad B, propagates. For example, if B expresses component failure and one of the arguments fails, the product will fail as well.

3. INTRODUCING QOS IN THE COMPOENT MODEL

QoS is introduced in the component calculus through the notion of a Q-algebra due to [4]. In brief, a *Q-algebra* is an algebraic structure $R = (C, \oplus, \otimes, \mathbin{⦶}, \mathbf{0}, \mathbf{1})$ such that $R_{\otimes} = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $R_{\mathbin{⦶}} = (C, \oplus, \mathbin{⦶}, \mathbf{0}, \mathbf{1})$ are both c-semirings. Intuitively, C is a QoS domain (*e.g.*, a measure of resource usage or availability) whereas $\oplus$ represents a *choice* between QoS values and $\otimes$ and $\mathbin{⦶}$, respectively, compose QoS values sequentially or concurrently.

As an example, we consider $(\mathbb{R}_+ \cup \{\infty\}, min, +, max, \infty, 0)$, where the QoS values are non-negative real numbers together with infinity, which can be used to specify the (shortest) time for component behavior performance. The additive operation is min, the sequential combininator is sum, and the concurrent combininator is max over the domain.

QoS information is included in the component model as an additional attribute: its execution generates a QoS value which is observable (*i.e.*, measurable). Formally, definition (1) changes to

$$\langle u_0 \in U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times C \times O)^I\rangle \tag{4}$$

where C is the domain of some Q-algebra $R = (C, \oplus, \otimes, \mathbin{⦶}, \mathbf{0}, \mathbf{1})$.

The definition of all component combinators change accordingly to take into account the need for equally combining the observed QoS levels of their parameters. For example, the dynamics of sequential composition becomes

$$\begin{aligned} a_{p;q} \;= & U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \\ & \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}(U_p \times C \times K) \times U_q \\ & \xrightarrow{\tau_r} \mathsf{B}(U_p \times C \times K \times U_q) \\ & \xrightarrow{\mathsf{B}(\mathsf{id}\times a_q)} \mathsf{B}((U_p \times C) \times \mathsf{B}(U_q \times C \times O)) \\ & \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times C \times (U_q \times C \times O)) \\ & \xrightarrow{\mathsf{BBa}^\circ} \mathsf{BB}((U_p \times C \times (U_q \times C)) \times O) \\ & \xrightarrow{\mu} \mathsf{B}((U_p \times C \times (U_q \times C)) \times O) \\ & \xrightarrow{\mathsf{B}(\mathsf{m}\times\mathsf{id})} \mathsf{B}((U_p \times U_q \times (C \times C)) \times O) \\ & \xrightarrow{\mathsf{B}(\mathsf{id}\times\otimes\times\mathsf{id})} \mathsf{B}(U_p \times U_q \times C \times O) \end{aligned}$$

4. CONCLUSIONS

This paper is a preliminary step in order to extend the component calculus documented in [2, 3] to deal, in a *systematic* way, with QoS constraints expressed through Q-algebras. It is expected that such an extension will enable formal reasoning about QoS-aware components on top of a precise coalgebraic semantics. The full development of the extended calculus is ithe topic of our current research.

5. REFERENCES

[1] J. Adamek. An introduction to coalgebra. *Theory and Applications of Categories*, 14(8):157–199, 2005.

[2] L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[3] L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. P. Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82.1, Warsaw, April 2003.

[4] T. Chothia and J. Kleijn. Q-automata: Modelling the resource usage of concurrent components. *Electronic Notes in Theoretical Computer Science*, 175(2):153–167, 2007.

[5] B. Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 237–280. Springer, 2002.

[6] D. A. Menascé. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, 2004.

[7] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[8] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.